



下载APP



16 | 划分土地（上）：如何划分与组织内存？

2021-06-14 LMOS

《操作系统实战45讲》

课程介绍 >



讲述：陈晨

时长 16:10 大小 14.81M



你好，我是 LMOS。

内存跟操作系统的关系，就像土地和政府的关系一样。政府必须合理规划这个国家的土地，才能让人民安居乐业。为了发展，政府还要进而建立工厂、学校，发展工业和教育，规划城镇，国家才能繁荣富强。

而作为计算机的实际掌权者，操作系统必须科学合理地管理好内存，应用程序才能高效稳定地运行。



内存管理是一项复杂的工作，我会用三节课带你搞定它。

具体我是这么安排的：这节课，我们先解决内存的划分方式和内存页的表示、组织问题，设计好数据结构。下一节课，我会带你在内存中建立数据结构对应的实例变量，搞定内存页的初始化问题。最后一节课，我们会依赖前面建好的数据结构，实现内存页面管理算法。

好，今天我们先从内存的划分单位讲起，一步步为内存管理工作做好准备。

今天课程的配套代码，你可以点击 [🔗 这里](#)，自行下载。

分段还是分页

要划分内存，我们就要先确定划分的单位是按段还是按页，就像你划分土地要选择按亩还是按平方分割一样。

其实分段与分页的优缺点，前面 MMU 相关的课程已经介绍过了。这里我们从内存管理角度，理一理分段与分页的问题。

第一点，从表示方式和状态确定角度考虑。段的长度大小不一，用什么数据结构表示一个段，如何确定一个段已经分配还是空闲呢？而页的大小固定，我们只需用位图就能表示页的分配与释放。

比方说，位图中第 1 位为 1，表示第一个页已经分配；位图中第 2 位为 0，表示第二个页是空闲，每个页的开始地址和大小都是固定的。

第二点，从内存碎片的利用看，由于段的长度大小不一，更容易产生内存碎片，例如内存中有 A 段（内存地址：0 ~ 5000）、B 段（内存地址：5001 ~ 8000）、C 段（内存地址：8001 ~ 9000），这时释放了 B 段，然后需要给 D 段分配内存空间，且 D 段长度为 5000。

你立马就会发现 A 段和 C 段之间的空间（B 段）不能满足，只能从 C 段之后的内存空间开始分配，随着程序运行，这些情况会越来越多。段与段之间存在着不大不小的空闲空间，内存总的空闲空间很多，但是放不下一个新段。

而页的大小固定，分配最小单位是页，页也会产生碎片，比如我需要请求分配 4 个页，但在内存中从第 1 ~ 3 个页是空闲的，第 4 个页是分配出去了，第 5 个页是空闲的。这种情

况下，我们通过修改页表的方式，就能让连续的虚拟页面映射到非连续的物理页面。

第三点，从内存和硬盘的数据交换效率考虑，当内存不足时，操作系统希望把内存中的一部分数据写回硬盘，来释放内存。这就涉及到内存和硬盘交换数据，交换单位是段还是页？

如果是段的话，其大小不一，A 段有 50MB，B 段有 1KB，A、B 段写回硬盘的时间也不同，有的段需要时间长，有的段需要时间短，硬盘的空间分配也会有上面第二点同样的问题，这样会导致系统性能抖动。如果每次交换一个页，则没有这些问题。

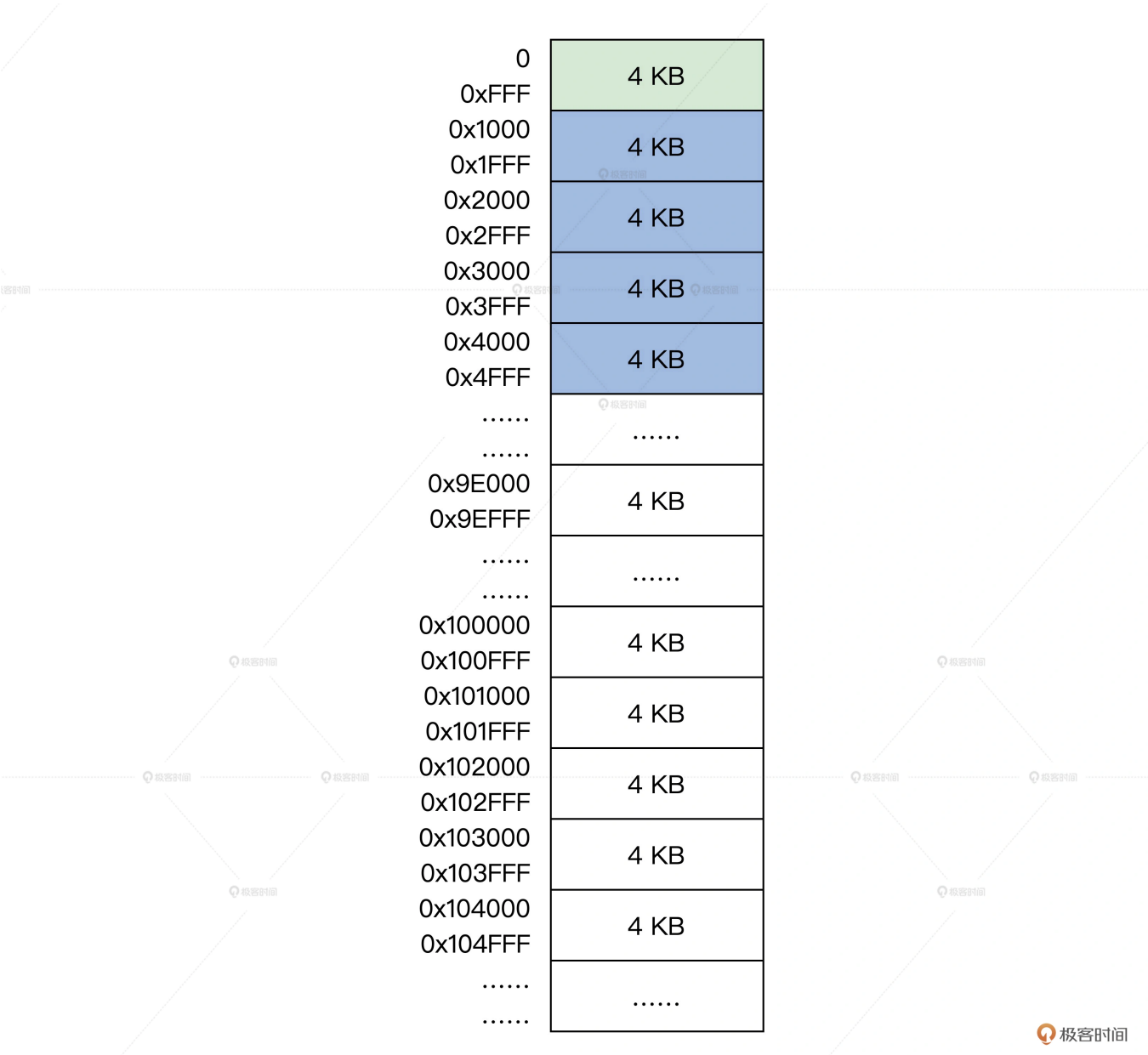
还有最后一点，**段最大的问题是使得虚拟内存地址空间，难于实施。**（后面的课还会展开讲）

综上，我们自然选择分页模式来管理内存，其实现在所有的商用操作系统都使用了分页模式管理内存。我们用 4KB 作为页大小，这也正好对应 x86 CPU 长模式下 MMU 4KB 的分页方式。

如何表示一个页

我们使用分页模型来管理内存。首先是把物理内存空间分成 4KB 大小页，这页表示从地址 x 开始到 $x+0xFFF$ 这一段的物理内存空间， x 必须是 $0x1000$ 对齐的。这一段 $x+0xFFF$ 的内存空间，称为**内存页**。

在逻辑上的结构图如下所示：



物理内存分页结构图

上图这是一个接近真实机器的情况，不过一定不要忘记前面的内存布局示图，真实的物理内存地址空间不是连续的，这中间可能有空洞，可能是显存，也可能是外设的寄存器。

真正的物理内存空间布局信息来源于 **e820map_t 结构数组**，之前的初始化中，我们已经将其转换成 **phymmap_t 结构数组**了，由 `kmachbsp->mb_e820expadr` 指向。

那问题来了，现在我们已经搞清楚了什么是页，但如何表示一个页呢？

你可能会想到位图或者整型变量数组，用其中一个位代表一个页，位值为 0 时表示页空闲，位值为 1 时表示页已分配；或者用整型数组中一个元素表示一个页，用具体数组元素的数值代表页的状态。

如果这样的话，分配、释放内存页的算法就确定了，就是扫描位图或者扫描数组。这样确实可以做出最简单的内存页管理器，但这也是最低效的。

上面的方案之所以低效，是因为我们仅仅只是保存了内存页的空闲和已分配的信息，这是不够的。我们的 Cosmos 当然不能这么做，我们需要页的状态、页的地址、页的分配计数、页的类型、页的链表，你自然就会想到，这些信息可以用一个 **C 语言结构体** 封装起来。

让我们马上就来实现这个结构体，在 cosmos/include/halinc/ 下建立一个 msadsc_t.h 文件，在其中实现这个结构体，代码如下所示。

[复制代码](#)

```

1 //内存空间地址描述符标志
2 typedef struct s_MSADFLGS
3 {
4     u32_t mf_olkty:2;    //挂入链表的类型
5     u32_t mf_lstty:1;    //是否挂入链表
6     u32_t mf_mocty:2;    //分配类型，被谁占用了，内核还是应用或者空闲
7     u32_t mf_marty:3;    //属于哪个区
8     u32_t mf_uindx:24;   //分配计数
9 }__attribute__((packed)) msadflgs_t;
10 //物理地址和标志
11 typedef struct s_PHYADRFLGS
12 {
13     u64_t paf_alloc:1;    //分配位
14     u64_t paf_shared:1;   //共享位
15     u64_t paf_swap:1;    //交换位
16     u64_t paf_cache:1;   //缓存位
17     u64_t paf_kmap:1;    //映射位
18     u64_t paf_lock:1;    //锁定位
19     u64_t paf_dirty:1;   //脏位
20     u64_t paf_busy:1;    //忙位
21     u64_t paf_rv2:4;     //保留位
22     u64_t paf_padr:52;   //页物理地址位
23 }__attribute__((packed)) phyadrflgs_t;
24 //内存空间地址描述符
25 typedef struct s_MSADSC
26 {
27     list_h_t md_list;     //链表
28     spinlock_t md_lock;   //保护自身的自旋锁
29     msadflgs_t md_inxflgs; //内存空间地址描述符标志
30     phyadrflgs_t md_phyadr; //物理地址和标志
31     void* md_odlink;      //相邻且相同大小msadsc的指针
32 }__attribute__((packed)) msadsc_t;

```

`msadsc_t` 结构看似很大，实则很小，也必须要小，因为它表示一个页面，物理内存页有多少就需要有多少个 `msadsc_t` 结构。正是因为页面地址总是按 4KB 对齐，所以 `phyadrflgs_t` 结构的低 12 位才可以另作它用。

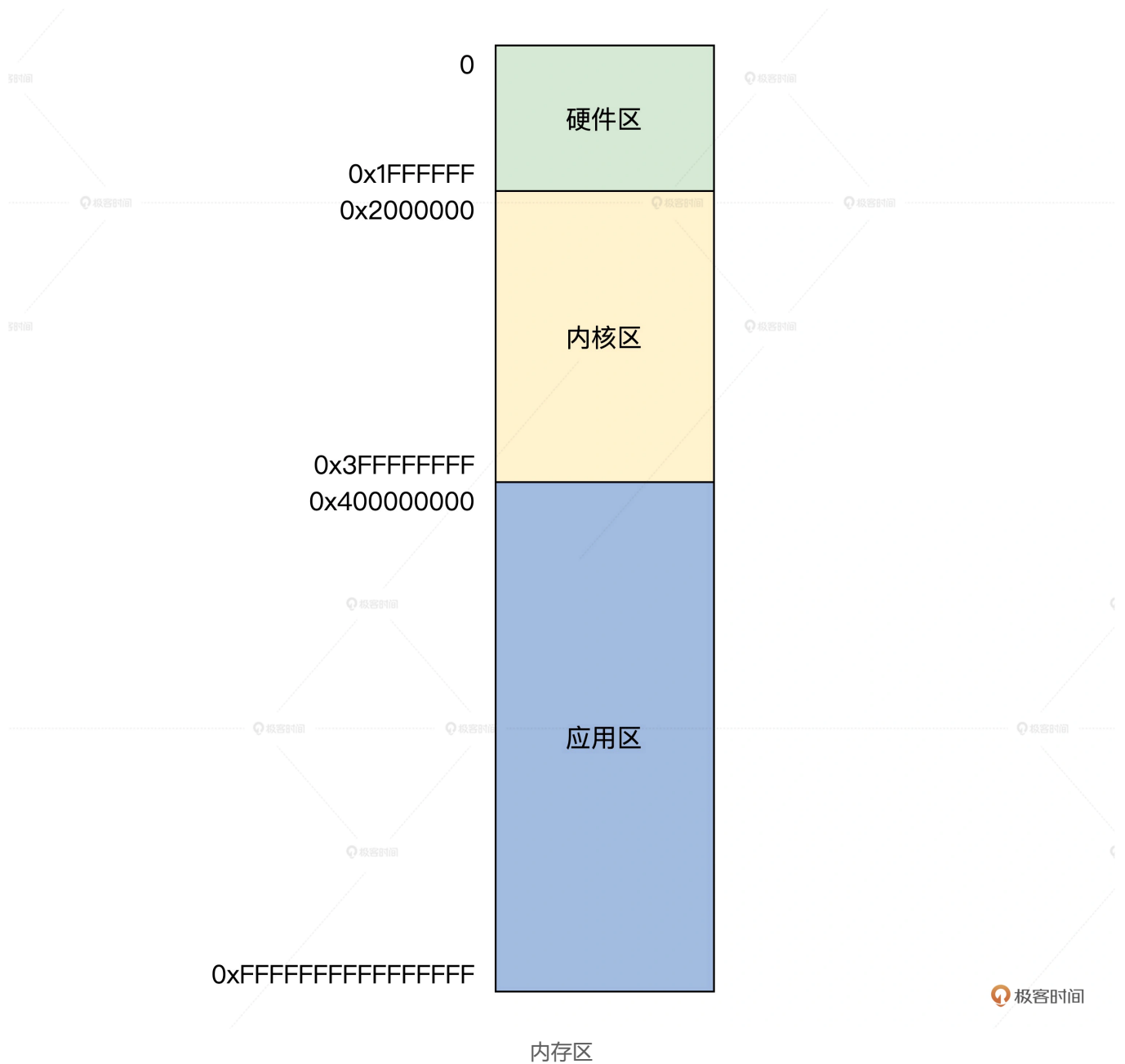
`msadsc_t` 结构里的链表，可以方便它挂入到其他数据结构中。除了分配计数，`msadflgs_t` 结构中的其他部分都是用来描述 `msadsc_t` 结构本身信息的。

内存区

就像规划城市一样，一个城市常常会划分成多个不同的小区，我们 Cosmos 的内存管理器不仅仅是将内存划分成页面，还会把多个页面分成几个内存区，方便我们对内存更加合理地管理，进一步做精细化的控制。

我想提醒你的是，内存区和内存页不同，内存区只是一个逻辑上的概念，并不是硬件上必需的，就是说就算没有内存区，也毫不影响硬件正常工作；但是没有内存页是绝对不行的。

那么内存区到底是什么？我们一起看一幅图就明白了，如下所示：



根据前面的图片，我们发现把物理内存分成三个区，分别为硬件区，内核区，应用区。那它们有什么作用呢？我们分别来看看。

首先来看**硬件区**，它占用物理内存低端区域，地址区间为 0~32MB。从名字就能看出来，这个内存区域是给硬件使用的，我们不是使用虚拟地址吗？虚拟地址不是和物理地址无关吗，一个虚拟可以映射到任一合法的物理地址。

但凡事总有例外，虚拟地址主要依赖于 CPU 中的 MMU，但有很多外部硬件能直接和内存交换数据，常见的有 DMA，并且它只能访问低于 24MB 的物理内存。这就导致了我们的很多内存页不能随便分配给这些设备，但是我们只要规定硬件区分配内存页就好，这就是硬件区的作用。

接着是**内核区**，内核也要使用内存，但是内核同样也是运行在虚拟地址空间，就需要有一段物理内存空间和内核的虚拟地址空间是线性映射关系。

再者，很多时候，内核使用内存需要大的、且连续的物理内存空间，比如一个进程的内核栈要 16KB 连续的物理内存、显卡驱动可能需要更大的连续物理内存来存放图形图像数据。这时，我们就需要在这个内核区中分配内存了。

最后我们来看下**应用区**，这个区域主要是给应用用户态程序使用。应用程序使用虚拟地址空间，一开始并不会为应用一次性分配完所需的所有物理内存，而是按需分配，即应用用到一页就分配一个页。

如果访问到一个没有与物理内存页建立映射关系的虚拟内存页，这时候 CPU 就会产生缺页异常。最终这个缺页异常由操作系统处理，操作系统会分配一个物理内存页，并建好映射关系。

这是因为这种情况往往分配的是单个页面，所以为了给单个页面提供快捷的内存请求服务，就需要**把离散的单页、或者是内核自身需要建好页表才可以访问的页面，统统收归到用户区。**

但是我们要如何表示一个内存区呢？和先前物理内存页面一样，我们需要定义一个数据结构，来表示一个内存区的开始地址和结束地址，里面有多少个物理页面，已经分配了多少个物理页面，剩下多少等等。

我们一起来写出这个数据结构，代码如下所示。

 复制代码

```
1 #define MA_TYPE_HWAD 1
2 #define MA_TYPE_KRNL 2
3 #define MA_TYPE_PROC 3
4 #define MA_HWAD_LSTART 0
5 #define MA_HWAD_LSZ 0x20000000
6 #define MA_HWAD_LEND (MA_HWAD_LSTART+MA_HWAD_LSZ-1)
7 #define MA_KRNL_LSTART 0x20000000
8 #define MA_KRNL_LSZ (0x40000000-0x20000000)
9 #define MA_KRNL_LEND (MA_KRNL_LSTART+MA_KRNL_LSZ-1)
10 #define MA_PROC_LSTART 0x40000000
11 #define MA_PROC_LSZ (0xffffffff-0x40000000)
12 #define MA_PROC_LEND (MA_PROC_LSTART+MA_PROC_LSZ)
13
```

```

14 typedef struct s_MEMAREA
15 {
16     list_h_t ma_list;           //内存区自身的链表
17     spinlock_t ma_lock;        //保护内存区的自旋锁
18     uint_t ma_stus;            //内存区的状态
19     uint_t ma_flg;            //内存区的标志
20     uint_t ma_type;            //内存区的类型
21     sem_t ma_sem;              //内存区的信号量
22     wait_l_head_t ma_waitlst;  //内存区的等待队列
23     uint_t ma_maxpages;        //内存区总的页面数
24     uint_t ma_alloppages;      //内存区分配的页面数
25     uint_t ma_freepages;       //内存区空闲的页面数
26     uint_t ma_resvpages;       //内存区保留的页面数
27     uint_t ma_horizline;       //内存区分配时的水位线
28     adr_t ma_logicstart;       //内存区开始地址
29     adr_t ma_logicend;         //内存区结束地址
30     uint_t ma_logicsz;         //内存区大小
31     //还有一些结构我们这里不关心。后面才会用到
32 } memarea_t;

```

好了，关于内存区的数据结构我们就设计好了，但是这仍然不能让我们高效地分配内存，因为我们没有把内存区数据结构和内存页面数据结构关联起来，如果我们现在要分配内存页依然要遍历扫描 `msadsc_t` 结构数组，这和扫描位图没有本质的区别。

下面我们就把它们之间关联起来，也就是组织内存页。

组织内存页

如何组织内存页呢？按照我们之前对 `msadsc_t` 结构的定义，组织内存页就是组织 `msadsc_t` 结构，而 `msadsc_t` 结构中就有个链表，你大概已经猜到了，我们组织 `msadsc_t` 结构正是通过另一个数据结构中的链表，将 `msadsc_t` 结构串连在其中的。

如果仅仅是这样，那我们将扫描这个链表，而这和之前扫描 `msadsc_t` 结构数组没有任何区别。

所以，我们需要更加科学合理组织 `msadsc_t` 结构，下面我们来定义一个挂载 `msadsc_t` 结构的数据结构，它其中需要锁、状态、`msadsc_t` 结构数量，挂载 `msadsc_t` 结构的链表、和一些统计数据。

 复制代码

```

1 typedef struct s_BAFHLST

```

```

2 {
3     spinlock_t af_lock;        //保护自身结构的自旋锁
4     u32_t af_stus;             //状态
5     uint_t af_oder;            //页面数的位移量
6     uint_t af_oderpnr;         //oder对应的页面数比如 oder为2那就是1<<2=4
7     uint_t af_fobjnr;          //多少个空闲msadsc_t结构，即空闲页面
8     uint_t af_mobjnr;          //此结构的msadsc_t结构总数，即此结构总页面
9     uint_t af_alcindx;          //此结构的分配计数
10    uint_t af_freindx;          //此结构的释放计数
11    list_h_t af_frelst;         //挂载此结构的空闲msadsc_t结构
12    list_h_t af_alclst;         //挂载此结构已经分配的msadsc_t结构
13 }bafhlst_t;

```

有了 bafhlst_t 数据结构，我们只是有了挂载 msadsc_t 结构的地方，这并没有做到科学合理。

但是，如果我们把多个 bafhlst_t 数据结构组织起来，形成一个 bafhlst_t 结构数组，并且把这个 bafhlst_t 结构数组放在一个更高的数据结构中，这个数据结构就是内存分割合并数据结构——memdivmer_t，那情况就不一样了。

有何不一样呢？请往下看。

 复制代码

```

1 #define MDIVMER_ARR_LMAX 52
2 typedef struct s_MEMDIVMER
3 {
4     spinlock_t dm_lock;        //保护自身结构的自旋锁
5     u32_t dm_stus;             //状态
6     uint_t dm_divnr;           //内存分配次数
7     uint_t dm_mernr;           //内存合并次数
8     bafhlst_t dm_mdmlielst[MDIVMER_ARR_LMAX]; //bafhlst_t结构数组
9     bafhlst_t dm_onemsa1st;    //单个的bafhlst_t结构
10 }memdivmer_t;
11

```

那问题来了，内存不是只有两个标准操作吗，这里我们为什么要用分割和合并呢？这其实取意于我们的内存分配、释放算法，对这个算法而言分配内存就是分割内存，而释放内存就是合并内存。

如果 memdivmer_t 结构中 dm_mdmlielst 数组只是一个数组，那是没有意义的。我们正是通过 dm_mdmlielst 数组，来划分物理内存地址不连续的 msadsc_t 结构。

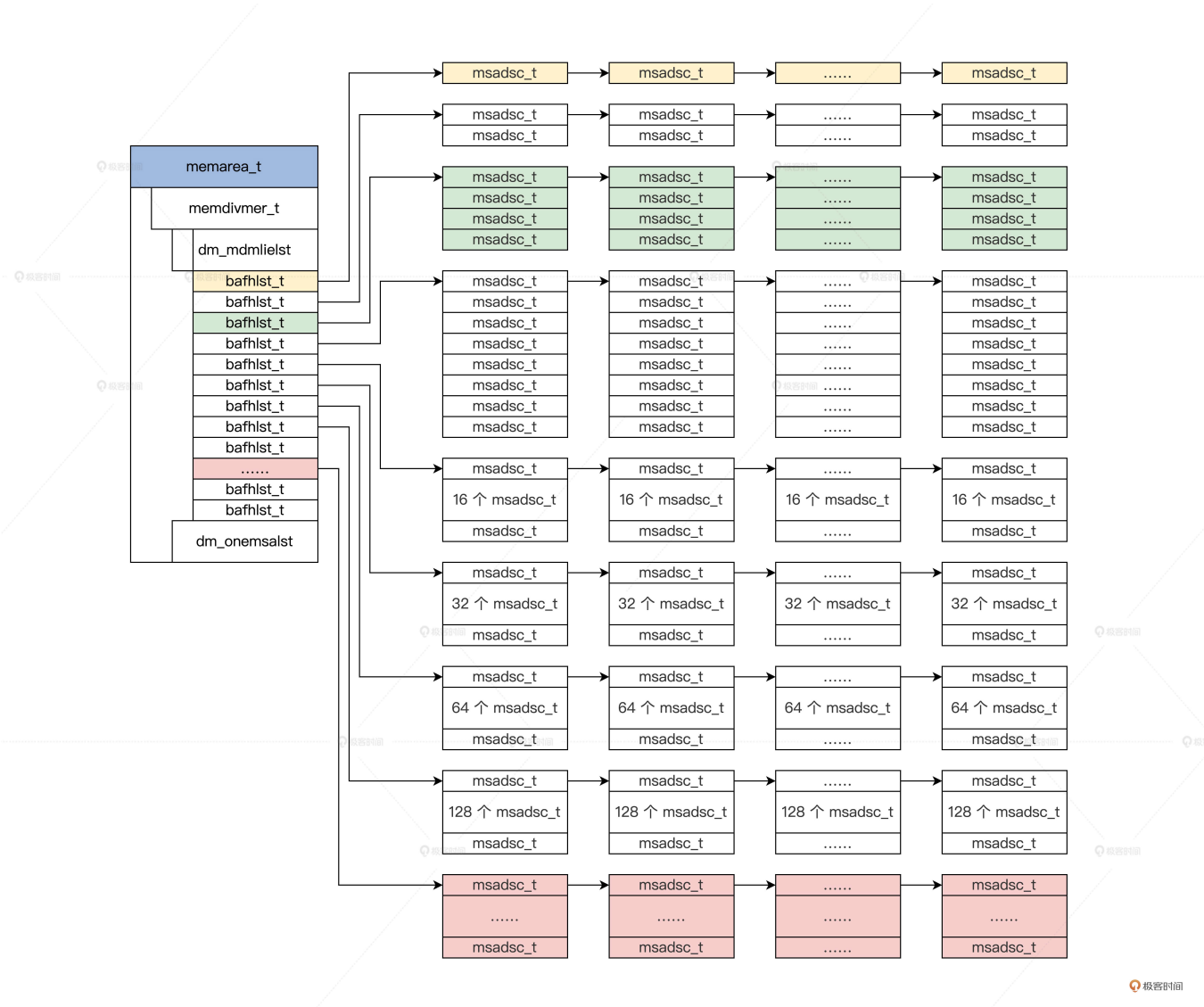
dm_mdmlielst 数组中第 0 个元素挂载单个 msadsc_t 结构，它们的物理内存地址可能对应于 0x1000，0x3000，0x5000。

dm_mdmlielst 数组中第 1 个元素挂载两个连续的 msadsc_t 结构，它们的物理内存地址可能对应于 0x8000 ~ 0x9FFF，0xA000 ~ 0xBFFF；dm_mdmlielst 数组中第 2 个元素挂载 4 个连续的 msadsc_t 结构，它们的物理内存地址可能对应于 0x100000 ~ 0x103FFF，0x104000 ~ 0x107FFF.....

依次类推，dm_mdmlielst 数组挂载连续 msadsc_t 结构的数量等于用 1 左移其数组下标，如数组下标为 3，那结果就是 8 ($1 << 3$) 个连续的 msadsc_t 结构。

需要注意的是，我们并不在意其中第一个 msadsc_t 结构对应的内存物理地址从哪里开始，但是第一个 msadsc_t 结构与最后一个 msadsc_t 结构，它们之间的内存物理地址是连续的。

如果还是不明白，我们来画个图就清楚了。



页面组织结构示意图

从上图上我们可以看出，每个内存区 `memarea_t` 结构中包含一个内存分割合并 `memdivmer_t` 结构，而在 `memdivmer_t` 结构中又包含 `dm_mdmlielst` 数组。在 `dm_mdmlielst` 数组中挂载了多个 `msadsc_t` 结构。

那么为什么要这么组织呢？后面我们在分配内存的时候，你就会明白了。

重点回顾

今天我们从比对分段与分页的区别开始思考，确定了使用分页方式，设计了内存页、内存区等一系列数据结构，下面我们来回顾一下本课程的重点。

1. 我们探讨了**分段与分页的区别**，发现段长度不一，容易产生内存碎片、不容易和硬盘换入换出数据，更不能实现扁平化的虚拟内存地址空间，由于这些不足我们选择了分页模式来管理内存，其实现在所有的商用操作系统都使用了分页模式管理内存。

2. 为了实现分页管理，首先是解决**如何表示一个物理内存页**，我们想到过位图和字节数组，但是它们遍历扫描，性能太差，于是设计了更复杂的 **msadsc_t 结构**，一个 msadsc_t 结构对应一个可用的物理内存页面。

3. 为了适应不同的物理地址空间的要求，比如有些设备需要低端的物理地址，而有的需要大而连续地址空间，我们对内存进行分区，**设计了 memarea_t 结构**。

每个 memarea_t 结构表示一个内存区，memarea_t 结构中包含一个内存分割合并 memdivmer_t 结构，而在 memdivmer_t 结构中又包含了 bafhlst_t 结构类型 dm_mdmlielst 数组。在 dm_mdmlielst 数组中挂载了多个 msadsc_t 结构。

思考题

我们为什么要以 2 的（0~52）次方为页面数来组织页面呢？

欢迎你在留言区跟我交流互动，也欢迎你把这节课分享给你的同事、朋友。

我是 LMOS，我们下节课见！

分享给需要的人，Ta订阅后你可得 **20 元现金奖励**

 赞 5  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 15 | Linux初始化（下）：从_start到第一个进程

下一篇 17 | 划分土地（中）：如何实现内存页面初始化？

更多学习推荐

极客时间 | 训练营

大厂面试必考 100 题

2021 最新版 | 算法篇

限量免费领取  仅限前 99 名

精选留言 (14)

 写留言**neohope** 置顶

2021-06-14

memarea_t , 进行内存区, 解决功能分区的问题
-> memdivmer_t , 进行内存分割合并管理
-> bafhlst_t , 以2的n次方对内存页面进行分组
-> msadsc_t , 解决单一页面管理问题

...

展开 

作者回复: 你好, 总结的相当到位



13

**pedro**

2021-06-14

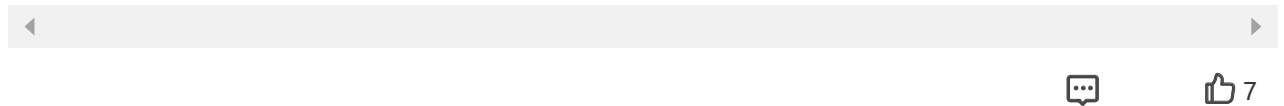
对于思考题, 原文中:

依次类推, dm_mdmlielst 数组挂载连续 msadsc_t 结构的数量等于用 1 左移其数组下标, 如数组下标为 3, 那结果就是 $8 (1 < 3)$ 个连续的 msadsc_t 结构。

因此页面数统统都是 2 的倍数, 8 是 2^3 , 这个地方我百思不得其解, 页面数为什么是 ...

展开 ∨

作者回复: 是的，这么操作 同时也是为了降低内存碎片



嗣树

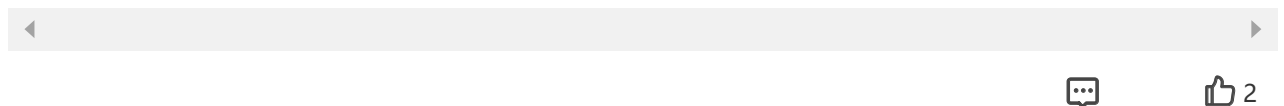
2021-06-25

neohope 老哥学习榜样，我做点补充吧。
之所以是 0-52 是因为 $64 - 12 = 52$ ，64位地址，12是页大小

而选用 2 的幂次可以把算术运算都转化为位操作，位操作是要比算术运算快的。应用层可能为了可读性而不去使用位操作，但是在内核中只要是需要性能都会往这方面靠，所以...

展开 ∨

作者回复: 是的



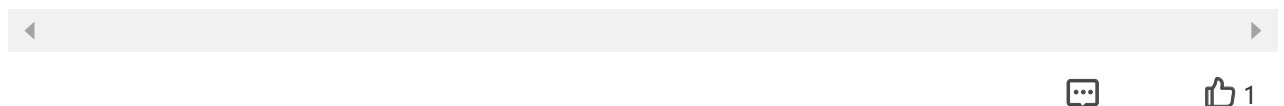
Fan

2021-06-27

思考题: 我们为什么要以 2 的 (0 ~ 52) 次方为页面数来组织页面呢？

应该是 内存分配时，根据需求大小快速定位至少从哪一部分开始

作者回复: 嗯嗯 猜对了



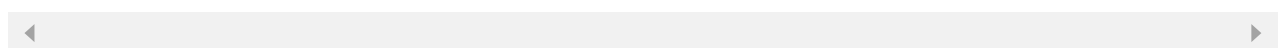
feihui

2021-06-19

因为计算机使用的是2进制，2的倍数可以表示任一大小，如： $3 = 4 - 1$ 如此高效分配，使用位移运算，假如用连续的自然数管理，在大内存申请的申请下需要一系列计算

展开 ∨

作者回复: 不完全 是哟





1

**Zhendicai**

2021-06-19

思考题应该是因为

dm_mdmlielst最后一个元素的下标是51也就是 2^{51} , $2^{51} \times 2^{12} = 2^{63}$, 就是说这时只需要两个bafhlst_t就能够表示完整的地址空间, 再多的话就超过了64位地址空间了。如果dm_mdmlielst再加一个元素的话 那就只需要一个bafhlst_t就能表示完整地址空间 但是没啥意义了...

展开 ▾

作者回复: 对的, 是这样的



1

1

**GroverZhu**

2021-06-14

这个内存分配很像C++ STL中早期用的allocator, 一个链表将可分配的内存串起来, 每个节点管理不同大小的内存块

作者回复: 这是管理 物理内存页面



1

**springXu**

2021-06-14

先留下个已阅的操作, 然后再分析内容。这一讲是对内存操作的规划图。要做到手中无图, 心里有图。 那算是学会了。

展开 ▾

作者回复: 对的, 正常的操作



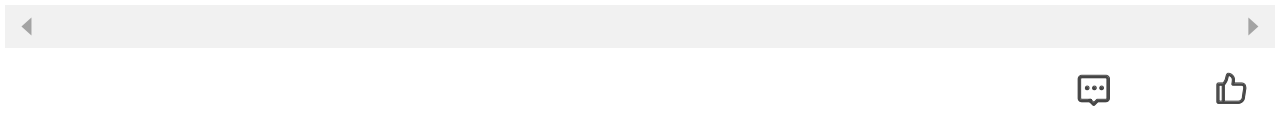
1

**吴建平**

2021-07-15

思考题: MDIVMER_ARR_LMAX定义为52, 表示从0~51页宽的bafhlst_t数组, 而2的0到51次方相加, 就等于2的52次方少1. 每页4k就是2的12次方。这样相当于, 每个bafhlst_t单元包含一个对应页宽的连续内存刚好等于64G空间少一个页面

作者回复: 2的52次方少1 个 连续的4KB页面



Mike_Han

2021-07-08

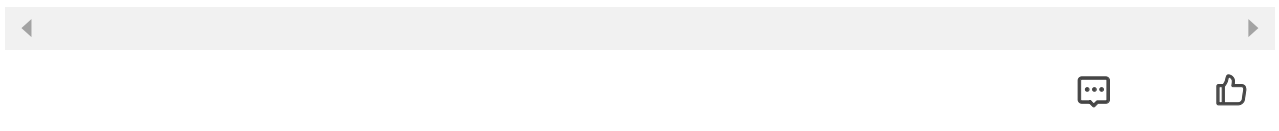
请教下各位大佬。

这个像是buddy，但是最大设置为2的52次方，这个太大了吧？有什么用呢？

好像buddy最多有12 ？2的12次方已经是4K个页面了，每个页面又有4K，这个就是连续内存最大16G了吧？那2的52次方个 4K 页面的连续空间，这个更大了吧？还是我的计算有问题？

展开 ∨

作者回复: 不是buddy



Mike_Han

2021-07-08

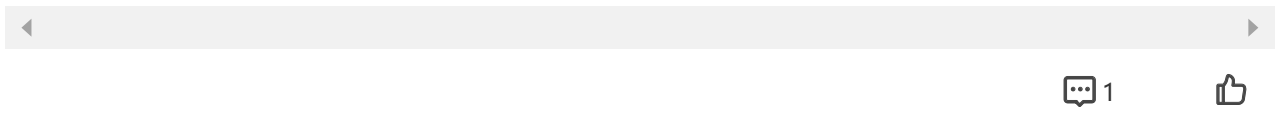
这是原文中的内容，我想请教个问题，

"需要注意的是，我们并不在意其中第一个 msadsc_t 结构对应的内存物理地址从哪里开始，但是第一个 msadsc_t 结构与最后一个 msadsc_t 结构，它们之间的内存物理地址是连续的。"

怎么做到"第一个 msadsc_t 结构与最后一个 msadsc_t 结构，它们之间的内存物理地址..."

展开 ∨

作者回复: 由代码控制的



Fan

2021-06-27

思考题 我们为什么要以 2 的 (0 ~ 52) 次方为页面数来组织页面呢？

思考题 还是不明白。这里以2 的 (0 ~ 52) 次方为页面数来组织页面，不会造成没办法组织完成整个页面吗？

展开 ▾

作者回复: 不会



! null
2021-06-25

最后像是buddy

展开 ▾

作者回复: 对的



thomas
2021-06-21

typedef struct s_BAFHLST 这个命名是什么的缩写？
比如msadsc_t 这个命名就有做说明 内存空间地址描述符（memory space address descriptor）

展开 ▾

作者回复: block alloc free head list